Objektorientierte Programmierung
Zusammenfassung

Maximilian Wolf

August 10, 2025

Contents

1 Grundlagen

Java ist eine eine streng typsichere, Imperative, Objekt-orientiere Hochprache.

1.1 Einordnung von Java

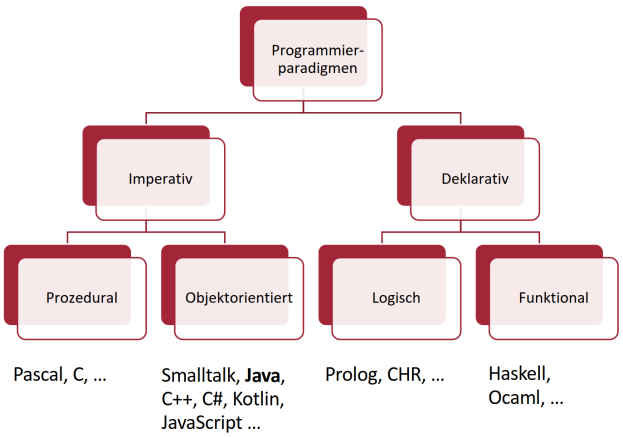

Figure 1: Einordnung von Java in Programmierparadigmen

1.2 Typisierung in Java

- **Typsicherheit** (verhindert Typsystem ungültige Operationen auf Werten?): Streng + implizite Konvertierung bei Zahlenbasistypen
- **Typisierung** (Explizit: Angabe von Datentypen im Code, Implizit: automatische Erkennung des Typs (Typinferenz)): Explizit, optional implizit
- **Typprüfung** (Statisch: während Kompilierzeit, Dynamisch: zur Laufzeit): Statisch + Dynamisch (insb. wegen Typcasts)
- **Typkompatibilität**: Nominal (Typen sind nur bei gleichem Namen bzw. gleicher Definition miteinander kompatibel)

1.3 Java Virtual Machine

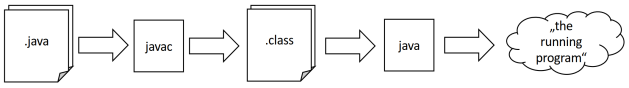

Figure 2: Java Compiler Spezifikation

1.4 Datentypen

1.4.1 Primitive Datentypen

- Werden direkt auf dem Stack gespeichert
- Zugriff per Call-by-Value
- Keine Klassen, jedoch existieren Wrapper-Klassen (Abbildung 4)
- **Autoboxing** Prim. Datentyp →Wrapper, **Unboxing** Wrapper →Prim. Datentyp

Typ	Größe	Format	Wertebereich
boolean	1 bit		true | false
char	16 bit	16-bit Unicode	16-bit-unicode-Zeichen
byte	8 bit	Zweierkomplement	-2^7 bis 2^7-1
short	16 bit	Zweierkomplement	-2^{15} bis $2^{15}-1$
int	32 bit	Zweierkomplement	-2^{31} bis $2^{31}-1$
long	64 bit	Zweierkomplement	-2^{63} bis $2^{63}-1$
float	32 bit	IEEE 754	binary32
double	64 bit	IEEE 754	binary64

Figure 3: Primitive Datentypen in java

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Figure 4: Wrapper Klassen zur Nutzung in Objektstrukturen

1.4.2 Referenztypen

- Werden im Heap gespeichert
- Zugriff per Referenz / Pointer
- "Call-by-value of the reference", d.h. kein echtes "Call-by-Reference", da Kopie der Referenz übergeben wird und die originale Referenz nicht verändert werden kann
- Garbage-Collection entfernt Objekt, sobald keine Referenz mehr existiert
- .equals() Methode zur Prüfung auf Gleichheit

1.5 Grundlegende Anweisungen

1.5.1 Ein-dimensionale Arrays

```
1  // Leeres Array mit 4 Elementen
2  String[] words = new String[4];
3
4  // Initialisierung mit 3 Werten
5  boolean[] sieve = new boolean[] {true, true, false};
```

1.5.2 Mehr-dimensionale Arrays

```
// Zwei-dimensionales Array
int[][] matrix = new int[][] {
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
};
```

1.5.3 Switch-Case

```
public static int fak(final int number) {
  switch (number) {
    case 0:
      return 1;
    default:
      return number * fak(number - 1);
  }
}
```

1.5.4 Iteration über Array

```
int[] xs = new int[] {1, 2, 3, 4, 5};

// For loop
for (int i = 0; i < xs.length; i++) {
  System.out.println(xs[i]);
}

// Enhanced for loop
for (int x : xs) {
  System.out.println(x);
}
```

1.6 Scope (Sichtbarkeitsbereich)

```
int a = 1;
{
  // Variable b nur in diesem Scope sichtbar
  int b = 2;
```

```
    System.out.println(b);
  }
```

2 Testing

2.1 JUnit-Tests

Example-based testing mit **assertEquals** Methode

```
class MathAlgorithmsTests {
  @Test
  void ggtTests() {
    assertEquals(12, MathAlgorithms.ggT(12, 36));
    assertEquals(1, MathAlgorithms.ggT(13, 19));
  }
}
```

2.2 jqwik-Tests

2.2.1 Automatische Generierung von Werten

Property-based testing mit **@ForAll**

```
class MathAlgorithmsProperties {

  // Property: ggT ist symmetrisch
  @Property
  boolean ggtIstSymmetrisch(@ForAll int a, @ForAll int
     b) {
    return MathAlgorithms.ggT(a, b) == MathAlgorithms.
       ggT(b, a);
  }

  // Property: ggT teilt beide Zahlen
  @Property
  boolean ggtTeiltBeide(@ForAll int a, @ForAll int b) {
    int g = MathAlgorithms.ggT(a, b);
    return g == 0 || (a % g == 0 && b % g == 0);
  }

  // Property: ggT(a, 0) == |a|
  @Property
```

```
18      boolean ggtMitNull(@ForAll int a) {
19        return MathAlgorithms.ggT(a, 0) == Math.abs(a);
20      }
21    }
```

2.2.2 Testen mit Arbitratries (Generatoren)

Property-based Testing mit Arbitraries (**@Provide**)

```
1   // Strings zum Testen
2   @Provide
3   Arbitrary<String> names() {
4     return Arbitraries.of("Ali", "Fatima", "Umar");
5   }
6
7   // Zahlenbereich
8   @Provide
9   Arbitrary<String> names() {
10    return Arbitraries.integers().between(0, 10);
11  }
12
13  // Methode zum Testen
14  @Property
15  void testNameIsNotNull(@ForAll("names") String name) {
16    Assertions.assertNotNull(name);
17  }
```

2.3 Logging

2.3.1 Log-Level

TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

2.3.2 Beispiel

```
1   public class Main {
2     private static Logger
3     logger = LogManager.getLogger(Main.class);
4     public static void main(String[] args) throws
          InterruptedException {
5       ...
6       logger.info(String.format("All workers started."));
```

```
7         ...
8         logger.debug(String.format("Sum of worker %d is %f.
              ", j, w.getSum()));
9         logger.trace(String.format("Currently accumulated
              sum is %f", sum));
10        ...
11        logger.info(String.format("All workers finished."))
              ;
```

3 Objekte und Klassen

3.1 Kapselung

- **Public** Zugriff von außen
- **Protected** Zugriff nur in selbem package bzw. in eigener Klasse oder vererbten Klasse
- **Private** Zugriff nur in eigener Klasse

3.2 Überladung

Angabe von mehreren Funktionen mit gleichem Namen und Rückgabewert in gleicher Klasse aber anderen Parametern
Virtuelle Methodenbindung Auswahl erfolgt zur Laufzeit nach Anzahl / Typen der Parameter

3.3 Vererbung

Keywort **extends**, Zugriff auf Superklasse durch **super**
Vererbte Klasse (Subklasse) erbt jede Methode der Superklasse
Annotation **@Override** zum Überschreiben von Methoden in der Subklasse

```
1   public class InheritingStack extends LList {
2     public void push(Object value) {
3       super.add (value);
4     }
5   }
```

3.3.1 Liskovsches Substitutionsprinzip

Methode der Subklasse verhält sich gleich wie die Superklasse

```
class Bird {
  public void fly() {
    System.out.println("Ich fliege");
  }
}

class Sparrow extends Bird {
  // OK: Sperling kann fliegen
}

// Ostrich ist zwar ein Bird, hat aber ein anderes
   Verhalten. Das widerspricht dem Prinzip
class Ostrich extends Bird {
  @Override
  public void fly() {
    throw new UnsupportedOperationException("Ein Strauß
        kann nicht fliegen!");
  }
}
```

3.4 Generics

```
public final class Optional<T> {
  private final T value;
  public boolean isPresent() {...}
  public boolean isEmpty() {...}
  public T get() {...}
  ...
}
```

3.4.1 Type Erasure

Da es Generics in Java erst seit Java 5 gibt, wurde Type Erasure eingefügt. Type Erasure bedeutet, dass Generics in Java zur Laufzeit gelöscht werden. Die Typen wie <String>, <Integer> usw. existieren nur zur Kompilierzeit, danach arbeitet die JVM mit Object.

```
Box<String> box = new Box<>();
box.set("Hallo");
```

```
3   String s = box.get();
4
5   // Nach Type Erasure wird daraus:
6   Box box = new Box();
7   box.set("Hallo");
8   String s = (String) box.get(); // Cast wird vom
        Compiler eingefügt
```

Dadurch entstehen folgende **Probleme**:

- Kein **instanceof** Box<String> möglich
- Keine Arrays mit generischen Typen (new T[10] nicht erlaubt)
- Gefahr von **ClassCastException** bei falscher Verwendung

3.5 Typecasting

3.5.1 Primitives Casting

- Casting zwischen *Primitiven Datentypen*
- **Widening Conversion**: Erfolgt automatisch, kein Datenverlust

```
1   int a = 5;
2   double b = a; // b = 5.0
```

- **Narrowing Conversion**: Datenverlust möglich

```
1   double b = 5.5;
2   int a = (int) b; // a = 5
```

3.5.2 Reference Casting

- **Upcasting**
  - Unterklasse →Oberklasse
  - Automatisch, weil jede Unterklasse eine Oberklasse ist
  - Kein Risiko

```
1   Dog d = new Dog();
2   Animal a = d; // Automatischer Upcast
```

- **Downcasting**

- Muss explizit mit *(Typ)* gemacht werden
- Compiler erlaubt es nur, wenn die Typen theoretisch kompatibel sind
- Laufzeitprüfung! →Falls das Objekt nicht wirklich der Zielklasse entspricht: *ClassCastException*

```
1  Animal a = new Dog();
2  Dog d = (Dog) a; // OK zur Laufzeit
3
4  Animal a2 = new Cat();
5  Dog d2 = (Dog) a2; // ClassCastException
```

3.6 Boxing

- **Autoboxing** Prim. Datentyp →Wrapper
- **Unboxing** Wrapper →Prim. Datentyp

3.7 Interfaces

Keyword **implements**
gibt vor, welche Methoden eine Klasse implementieren muss, ohne diese selbst zu implementieren. Siehe außerdem Functional Interfaces (5.2)

```
1  public interface Person() {
2    String getName();
3    int getAge();
4    ...
5  }
6
7  public class Paul implements Person() {
8    @Override
9    public String getName() { return "Paul"; }
10
11   @Override
12   public int getAge() { return 24; }
13
14   ...
15 }
```

3.8 Abstrakte Klassen

Keyword **abstract**
Teilimplementierung von Methoden, keine Instanziierung möglich

```
public abstract class doStuff {
  private final int n = 5;
  // Implementierung einer Methode
  public int getN() { return this.n };

  // Vorgabe einer Methode
  String toText(int a);
}
```

3.9 instanceof

Prüft, ob Objekt Instanz von einem Interface, Objekt oder Subklasse ist

```
List xs = new ArrayList();
xs instanceof List ==> true
```

3.10 Wildcards

Generische und sichere Möglichkeit mit Generics zu arbeiten

- `<?>` Wildcard akzeptiert jeden Typ
- `<? extends T>` (**Upper bound** Wildcard) akzeptiert nur Typ, der T ist oder davon erbt
- `<? super T>` (**Lower bound** Wildcard) akzeptiert nur Typ, der T ist oder ein Supertyp von T ist

```
List<?> liste

// Upper bound Wildcard
List<? extends Number> zahlen;

// Lower bound Wildcard
List<? super Integer> liste;

<R> Stream<R> map(Function<? super T, ? extends R>
   mapper);
```

3.11 Enumerations

Aufzählungstypen können auch Body mit Konstruktoren, Attributen und Methoden haben

```
public enum StandardOpenOption implements OpenOption {
  READ,
  WRITE,
  APPEND,
  ...
}

public enum GeldStuecke {Cent(1), Cent_2(2), Cent_5(5),
    Cent_10(10), Cent_20(20), Euro_1(100), Euro_2(200);
  private int cent_value;
  private GeldStuecke(int value) {
    this.cent_value = value;
  }
  public int getCent_Value() {
    return cent_value;
  }
}
```

3.12 Records

Datenkonstrukt, dass automatisch Object Methods (z.B. equals()) und getter-Methoden erzeugt, Attribute sind final (immutable)

```
public record Haus(String strasse, int hausnummer) {}

Haus haus = new Haus("Jahnstrasse", 5);
System.out.println(haus.strasse()); // "Jahnstrasse"
```

4 Collections

Collection<E> ist das Basis-Interface für alle **Single-Value**-Strukturen, Subklasse von Iterable<E>, d.h. alle Klassen, die von Collection<E> erben, sind iterierbar

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object o);
```

```
5     boolean add(E e);
6     boolean remove(Object o);
7     boolean removeIf(Predicate<? super E> filter);
8     ...
9   }
```

4.1 Queue Interfaces

4.1.1 Queue<E>

Standard FIFO Queue
Implementierungen LinkedList<E>, PriorityQueue<E>, etc.

```
1   public interface Queue<E> extends Collection <E> {
2     boolean add(E e);
3     boolean offer (E e);
4     E peek();
5     E remove();
6     E poll();
7     E remove();
8   }
```

4.1.2 Deque<E>

Queue, die Einfügen und Herausnehmen sowohl *vorne* als auch *hinten* unterstützt
Implementierungen ArrayDeque<E>

4.1.3 BlockingQueue<E>

Blockiert beim Einfügen oder Herausnehmen von Elementen, nützlich bei nebenläufigen Programmen, die Synchronisation erforderlichen
Implementierungen LinkedBlockingQueue<E>

4.2 Map und Set Interfaces

4.2.1 Map<K,V>

Speichert **Key-Value**-Paare
Implementierungen HashMap<E>, TreeMap<E>, etc.

```
1   public interface Map<K, V> {
2     ...
```

```
3     V get(Object key);
4     V put(K key, V value);
5     V remove(Object key);
6     boolean containsKey(Object key);
7     boolean containsValue(Object value);
8     ...
9   }
```

4.2.2 Set<E>

Speichert Elemente eindeutig
Implementierungen HashSet<E>, TreeSet<E>, etc.

4.3 Überblick

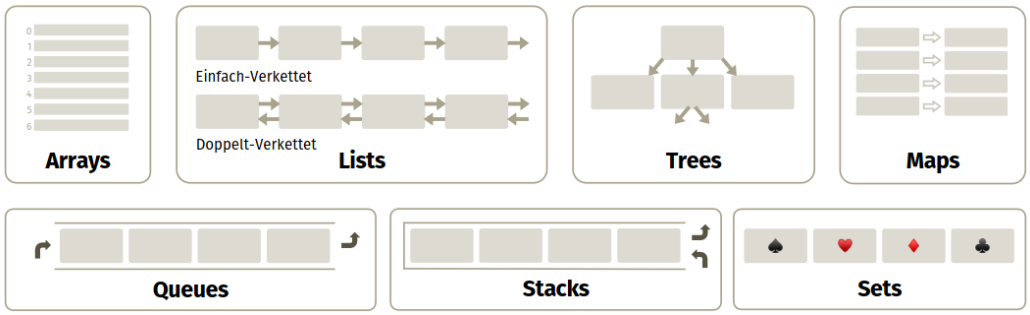

Figure 5: Collections Datentypen in Java

5 Funktionale Programmierung

5.1 Lambda Expressions

Eine Lambda Funktion hat beliebige Anzahl an Parametern und einen einzigen Output, eine beispielhafte Implementierung einer λ-Funktion sehe so aus:

```
1   BinaryOperator<Integer> mult =
2   (Integer a, Integer b) -> {
3     return a * b;
4   };
5   System.out.println(mult.apply(21, 2));
```

5.2 Functional Interfaces

Ein Functional Interface ist ein Interface (Unterabschnitt 3.7), darf und muss nur eine abstrakte Methode implementieren. Ein Functional Interface wird mit dem Keyword **@FunctionalInterface** deklariert.

```
1  @FunctionalInterface
2  public interface MyFunction {
3    int apply(int x, int y);
4  }
```

5.3 Stream-API

Ein Stream ist im Gegensatz zu einer Collection keine Datenstruktur, sondern eine Berechnungspipeline für Elemente einer Quelldatenstruktur. Dabei wird jedes Element in der Pipeline nur einmal verarbeitet.

5.3.1 Objekt-Streams

```
1  Stream<String> stringStream = Stream.of("a", "b", "c");
```

5.3.2 Primitive Streams

Spezielle Methoden: *sum()*, *average()*, *max()*, *min()*, *range()*, *rangeClosed()*

- **IntStream** (int)
- **LongStream** (long)
- **DoubleStream** (double)

Beispiele:

```
1  IntStream.range(1, 5); // 1 - 4 (int)
2  LongStream.range(1, 5); // 1 - 4 (long)
```

5.3.3 Array-Streams

Erzeugt einen **Objekt-Stream** (5.3.1) mit dem Typen (Generic) des Arrays

```
1  int[] values = new int[] {1, 2, 3, 4};
2  Arrays.stream(values);
```

5.3.4 Unendliche Streams

→**Lazy-Evaluation** (5.3.7) **.limit(...)** zur Begrenzung der Werte

```
Stream.generate(...);
Stream.iterate(...);
new Random().ints();
```

5.3.5 Parallele Streams

- Keyword **.parallelStream()**
- Nebenläufige Verarbeitung von Streams
- Operationen müssen Stateless (5.3.9) und non-interfering(5.3.10) sein

```
public static int sumOfSmalls(List<Integer> values) {
  return values.parallelStream()
  .filter(x -> x < 3)
  .reduce(0, (a, b) -> a + b);
}
```

5.3.6 Ordnung

Ein Stream ist geordnet, wenn eine definierte relative Reihenfolge der Elemente besteht und beibehalten wird

- Geordnet z.B. **.stream()**
- Ungeordnet **.unordered()**, erlaubt der JVM eine effizientere Bearbeitung von parallelen Streams (5.3.5)

5.3.7 Lazyness

Lazyness bedeutet, nur das nötigste zu tun und zum spätestmöglichen Zeitpunkt

```
public static void streaming(List<String> values) {
  var list = values.stream()
  .map(String::toUpperCase)
  .filter(x -> x.length() > 3)
  .skip(3) // Stream ohne ersten 3 Elemente
  .toList();
  System.out.println(list);
}
```

5.3.8 Intermediate vs. Terminal Operations

- **Intermediate** Operationen erzeugen neue Streams aus Streams und sind lazy
- **Terminal** Operationen produzieren direkt oder über Seiteneffekte ein Ergebnis, sind nicht lazy, sondern eager und beenden den Stream

5.3.9 Stateless vs. Stateful Operations

- **Stateless** Operationen speicher keinen Zustand in Bezug auf vorherige Elemente, jedes Element kann einzeln verarbeitet werden z.B. filter / map
- **Stateful** Operationen benötigen Informationen über vorherige Elemente, z.B. distinct / sorted

5.3.10 Interfering vs. Non-interfering

Ein Stream ist **non-interfering**, wenn die zugrunde liegende Datenquelle nicht verändert wird. so z.B.

```
1  List<String> names = List.of("Ali", "Fatima", "Umar");
2  names.stream().forEach(System.out::println);
```

Ein Stream ist **interfering**, wenn die Datenquelle während des Stream-Prozesses verändert wird (z.B. durch Seiteneffekte)

```
1  List<String> names = new ArrayList<>(List.of("Ali", "
       Fatima", "Umar"));
2
3  names.stream().peek(name -> {
4    if (name.equals("Ali")) {
5      names.remove(name); // Element wird aus Stream
           entfernt
6    }
7  }).forEach(System.out::println);
```

5.3.11 Seiteneffekte

Eine Funktion hat **Seiteneffekte**, wenn sie Zustände außerhalb von sich selbst ändert. Beispiel:

```
1  // Seiteneffektfrei
2  int square(int x) {
3    return x * x;
4  }
5
6  // Variable wird global verändert -> Seiteneffekte
7  int counter = 0;
8  int next() {
9    counter++;
10   return counter;
11 }
```

5.4 Typen und Interfaces

5.4.1 Predicate<T>

Prüfung von Elementen

5.4.2 BinaryOperator<T>

2 Operanden, 1 Operator und ein Rückgabetyp, alle vom Typ T

5.4.3 BiFunction<T,U,R>

2 Operanden, 1 Operator und ein Rückgabetyp, alle haben verschiedene Typen

5.4.4 Function<T,R>

1 Operand und ein Rückgabetyp, unterschiedliche Typen

5.4.5 Consumer<T>

1 Operand und kein Rückgabetyp, operiert per Seiteneffekte (5.3.11)

5.5 Intermediate Stream Operations

- **.filter(Predicate)** filtert Elemente heraus, auf die das Prädikat nicht zutrifft

- **.map(Function)** erlaubt das Verändern von Elementen aus einem Stream
- **.distinct()** entfernt mehrfach vorkommende Elemente
- **.sorted()** sortiert den Stream nach Standard-Sortierung, bzw. **.sorted(Comparator)** nach benutzerdefinierter Sortierung
- **.peek(Consumer)** erlaubt das Einsehen von Elementen
- **.limit(n)** gibt nur die ersten n Elemente weiter
- **.skip(n)** Überspringt die ersten n Elemente

5.6 Terminal Stream Operations

- **.forEach(Consumer)** wendet eine Aktion auf jedes Element an
- **.toArray()** gibt den Stream als Array zurück
- **.collect(Collector)** überführt Elemente zu einer Collection (z.B. List)
- **.count()** zählt die Anzahl an Elementen
- **.anyMatch(Predicate)** prüft, ob mindestens ein Element das Prädikat erfüllt
- **.allMatch(Predicate)** prüft, ob alle Elemente das Prädikat erfüllen
- **.noneMatch(Predicate)** prüft, ob kein Elemente das Prädikat erfüllt
- **.findFirst()** gibt das erste Element als Optional aus
- **.findAny()** gibt irgendein Element als Optional zurück

5.6.1 Reduktion (Fold)

.reduce(start, acc, combiner) reduziert einen Stream auf ein einzelnes Ergebnis (terminal) 5.3.8, wobei mit einem Startwert das Ergebnis akkumuliert wird

```
1  public static int generalReduceSum(List<Integer> values
       ) {
2    BiFunction<Integer, Integer, Integer> accumulator = (
       a, b) -> a + b;
3    BinaryOperator<Integer> combiner = (r1, r2) -> r1 +
       r2;
4    return values.stream().reduce(0, accumulator,
       combiner);
5  }
```

Mutierende Reduktion
→erzeugt Reduktion durch Änderung des Containers

```
1  public static String concatStringsViaStringBuffer(List<
       String> strings) {
2    return strings
3    .stream()
4    .collect(StringBuffer::new,
5    StringBuffer::append,
6    StringBuffer::append)
7    .toString();
8  }
9  public static String concatStringsViaStringCopies(List<
       String> strings) {
10   return strings
11   .stream()
12   .reduce("", String::concat);
13 }
```

6 Input / Output (IO)

6.1 Zeichenkodierungen

- ASCII - 7 Bit, Zeichensatz 128 Zeichen
- ISO-8859-1 (Latin-1) - 8 Bit, 256 Zeichen (darunter z.B. äöüß)
- UTF-8 1-4 Byte, alle Unicode Zeichen, kompatibel mit ASCII

- UTF-16 2-4 Byte, viele Sprachen, nicht ASCII-kompatibel

6.2 Dateien

6.2.1 Dateiformate

- Textdatei: Codierung, Zeilenende - menschenlesbar
- Binärdatei: Endianess, Format, Dateiformatspezifikation - maschinenlesbar

6.3 Dateizugriff in Java

6.3.1 Datenflüsse (Streams)

Interfaces

- **InputStream** - read, skip, reset, close
- **OutputStream** - write, flush, close

Dateizugriffe

- **Byte Streams**: binäre Rohdaten
- **File Streams**: Lese- / Schreiboperationen auf Dateien
- **Buffered Streams**: Puffert die Schreib- und Leseoperationen
- **Character Streams**: Textdateien (inkl. Encoding)
- **Standard-I/O**: Standardein- und -ausgabe von der Konsole
- **Data Streams**: binäre Datenströme von Basisdatentypen und Strings
- **Object Streams**: binäre Datenströme von beliebigen Objekten

6.4 Exceptions

```
1  try {
2    // Try Block (Happy Path)
3  } catch (Exception e) {
4    // Catch Block (Fehlerbehandlung)
5  } finally {
6    // Finally Block - wird immer ausgeführt
7  }
```

6.4.1 Checked Exceptions

```
1  public void function() throws Exception {
2    // Zur Kompilierzeit muss Fehlerbehandlung vorhanden
       sein
3  }
```

6.4.2 Try-With-Resource-Statement

Vorraussetzung: AutoCloseable Interface

```
1  try (FileInputStream in = new FileInputStream("in.txt")
       ;
2    FileOutputStream out = new FileOutputStream("out.txt"
       )) {
3      ...
4    }
5  }
```

6.4.3 Decorater Pattern

Flexible Erweiterung eines Objekts zur Laufzeit, ohne die Klasse zu verändern.

- **Komponente** (Interface): Gemeinsame Schnittstelle
- **Konkrete Komponente**: Das ursprüngliche Objekt
- **Decorator** (Abstrakt): Implementiert die gleiche Schnittstelle und enthält eine Referenz auf ein anderes Objekt der Schnittstelle
- **Konkreter Decorator**: Erweitert das Verhalten

Beispiel

```
1  new BufferedInputStream(
2    new FileInputStream("Datei.txt")
3  );
```

6.5 Formatierte Textausgabe

%[argument_ index$][flags][width][.precision]conversion
→Argumente in eckigen Klammern sind optional

- argument_index: Position in der Argumentliste, Abkürzung "<" für vorhergehendes Argument
- flags: z.B. führende Nullen, Vorzeichen, Linksbündigkeit
- width: minimale Breite in der das Argument gesetzt wird
- precision: Anzahl der Nachkommastellen
- conversion: Formatierungshinweise

6.6 Java New I/O (NIO)

Wichtige Klassen

- FileSystem(s): Wurzelverzeichnisse / Arten von Dateisystemen
- Path(s): Pfade zu Verzeichnissen oder Dateien
- Files: copy, create, move, delete, attributes, owner,

Methoden

- **readAllBytes, readAllLines**: kleine Dateien
- **newBufferedReader, newBufferedWriter**: Textdateien
- **newInputStream, newOutputStream**: ungepufferte Streams benutzbar mit alter API
- **newByteChannel**: Kanäle und ByteBuffer
- **FileChannel**: Advanced Features, Dateisperren, Memory-mapped IO

Java IO	JAVA NIO
Streams	Buffers
Blockierende I/O	Nicht-blockierende I/O
-	Selektoren (warten auf mehrere Kanäle gleichzeitig)

6.7 Scanner

Scanner zerlegen Fließtext in logische Einheiten (Tokens)

```
Scanner(File source)
Scanner(File source, String charsetName)
Scanner(InputStream source)
Scanner(InputStream source, String charsetName)
Scanner(String source)
boolean hasNext() //Standardtrenner: Whitespaces
boolean hasNextDouble()
boolean hasNextInt()
boolean hasNextBoolean()
```

6.8 Regular Expression (Regex)

- Konkrete Zeichen: a b c A @ 0
- Beliebiges Zeichen: .
- Zeichenauswahl: [a-zA-Z0-9-+_]
- (a)? 0- oder 1-mal pattern a
- (a)* beliebig oft pattern a
- (a)+ mind. 1-mal
- \. (Escape durch \)
- \\ : \
- ^ : Beginn
- $: Ende

6.8.1 Capture Groups

Gruppennummierung
Capture groups (0 = gesamter Ausdruck)

```
(  ( ) ( ( ) ( ) ) ) ( )
1   2   3 4   5       6
```

Beispiel

(\d)	Ziffer
(\w)	Wortzeichen
(\b)	Wortgrenze
(\s)	Non-Whitespace
([a-z]{n})	n-mal a-z
([a-z]{n,p})	n bis p-mal a-z

Table 1: Capture Group Regex Syntax

```java
import java.util.regex.*;

String text = "User: Anna, Alter: 22";
Pattern p = Pattern.compile("User: (\\w+), Alter: (\\d+)");
Matcher m = p.matcher(text);

if (m.find()) {
  System.out.println("Name: " + m.group(1));  // Anna
  System.out.println("Alter: " + m.group(2)); // 22
}
```

6.9 XML

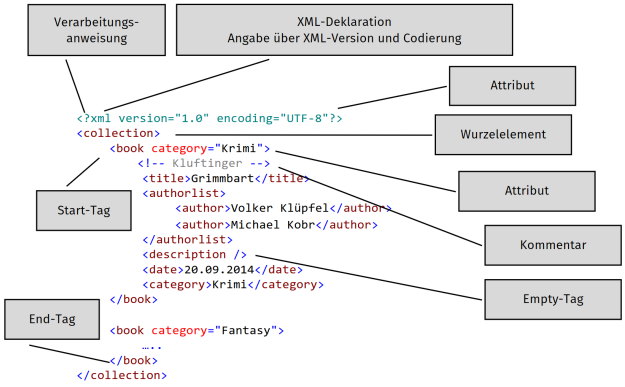

Figure 6: Aufbau eines XML-Dokuments

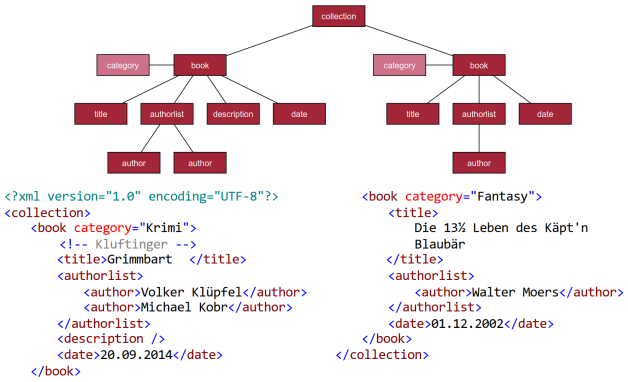

Figure 7: XML-Dokument als Baumstruktur

6.9.1 Syntaxregeln

<	<
>	>
&	&
'	'
"	"

Table 2: Syntaxregeln für belegte Zeichen in XML

6.9.2 Namespaces

Problem: Tags wie "Name" werden häufig wiederverwendet →Namespaces helfen, einen Tag zu einem Namespace hinzuzufügen.
Mit *xmlns* werden Namespaces definiert

```
<collections xmlns:b="Bücher" xmlns:dvd="DVDs">
  <b:collection>
    <book category="Krimi">
    ..
    </book>
    <book category="Fantasy">
    ..
  </b:collection>
  <dvd:collection>
    <dvd genre="Tierfilme">
    ..
    </dvd>
  </dvd:collection>
</collections>
```

6.10 Wohlgeformtes vs. Gültiges XML

- *Wohlgeformt*: Syntax korrekt
- *Gültig*: Struktur entspricht vorgebenem Schema
- *Schema*: natürlichsprachlich vs. formale Schemasprache (DTD, XSD)

6.11 Document Type Definition (DTD)

Verweis auf DTD in XML-Datei:

```
<!DOCTYPE collection SYSTEM "booklist.dtd">
```

Inhalt von booklist.dtd:

```
<!ELEMENT collection (book+)>
<!ELEMENT book (title, authorlist,
description?, date)>
<!ATTLIST book category CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT authorlist (author+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT description EMPTY>
<!ELEMENT date (#PCDATA)>
```

Probleme mit DTD nicht möglich ist:

- Anzahl der Instanzen angeben
- Aussehen der Zeichendaten innerhalb eines Elements spezifizieren
- Semantische Bedeutung eines Elements

6.12 XML Schema Definition (XSD)

6.12.1 Definition und Referenz

booklist.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="bookcollection"
        xmlns="bookcollection"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
..
</xs:schema>
```

booklist.xml

```
<collection
  xmlns="bookcollection"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="bookcollection booklist.xsd">
  ..
</collection>
```

6.12.2 Simple Types

- **Element**

```
<xs:element name="author" type="xs:string"/>
```

- **Attribut**

```
<xs:attribute name="category" type="xs:string"/>
```

- **Default-Wert**

```
<xs:attribute name="category" type="xs:string"
   default="unsorted"/>
```

- **Required**

```
<xs:attribute name="category" type="xs:string" use=
   "required"/>
```

6.12.3 Einschränkungen

- **Wertebereich**: minInclusive, maxInclusive, minExclusive, maxExclusive
- **Regex**: pattern
- **Längen**: length, maxLength, minLength
- **Ziffern**: fractionDigits, totalDigits
- **Whitespace**
  - **preserve**: Whitespaces bleiben erhalten
  - **replace**: Tabs, Zeilenumbrüche etc. werden zu Leerzeichen

– **collapse**: wie replace, jedoch auch bei mehreren

```
<xs:element name="speed">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="350"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Named Types und Referenz

```
<xs:simpleType name="speedType">
  <xs:restriction base="xs:int">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="350"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="speed" type="speedType"/>
```

Enumeration

```
<xs:element name="RegierungsbezirkeBW" type="regBWType"
   />
<xs:simpleType name="regBWType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Freiburg"/>
    <xs:enumeration value="Karlsruhe"/>
    <xs:enumeration value="Stuttgart"/>
    <xs:enumeration value="Tübingen"/>
  </xs:restriction>
</xs:simpleType>
```

6.12.4 Complex Types

Mögliche Kindelemente von *<complexType>*:

- *sequence*: alle in der angegebenen Reihenfolge
- *choice*: entweder-/ oder
- *all*: alle in beliebiger Reihenfolge
- Bei keiner Angabe: leeres Element

6.12.5 Data Types

string, normalizedString (ohne Leerzeichen an Rändern), int, byte, long, double, float, boolean, time, data, ID, IDREF(s), NMTOKEN(s), etc.

6.13 XPath

Pfadausdrücke zur Navigation in XML-Dokumenten

6.13.1 Pfadausdrücke

Symbol	Bedeutung
/	root-Node
<node-name>	alle Knoten mit diesem Namen
.	aktueller Knoten
..	parent-Knoten
//	alle (Sub-)Knoten ab dem aktuellen Knoten
@	Selektion von Attributen

Table 3: XPath Expressions

6.13.2 Prädikate

Symbol	Bedeutung
[1]	erster Kindknoten
[last()]	letzter Kindknoten
[last()-1]	vorletzter Kindknoten
[position()<3]	die ersten zwei Kindknoten
[@category]	alle Knoten mit Attribut `"category"`
[@category='Krimi']	alle Knoten, deren Attribut `"category"` den Wert `"Krimi"` hat
[@price > 35.00]	alle Knoten, deren **Element** `"price"` größer als 35.00 ist
[starts-with(@type, "tree")]	alle Knote, deren `"type"`-Attribut mit "tree" beginnt

6.13.3 Verwendung in Java

```
1  XPathFactory xPathFactory = XPathFactory.newInstance();
2  XPath xpath = xPathFactory.newXPath();
3  XPathExpression expr = xpath.compile(<XPath expression
      >);
```

```
4   NodeList nodes = (NodeList) expr.evaluate(doc,
        XPathConstants.NODESET);
```

6.14 DOM (Document Object Model)

```
1   DocumentBuilderFactory factory = DocumentBuilderFactory
        .newDefaultInstance();
2   DocumentBuilder builder = factory.newDocumentBuilder();
3   Document doc = builder.parse(new File(path));
4
5   NodeList nodeList = doc.findElementsByTagName("truck");
```

6.15 JSON (Javascript Object Notation)

```
1   Gson gson = new Gson();
2   try (FileReader reader = new FileReader(path)) {
3     Obj o = gson.fromJson(reader, o.class);
4   } catch (Exception e) {...}
```

```
1   JsonElement root = JsonParser.parseReader(new
        FileReader(path));
2   JsonObject rootObj = root.getAsJsonObject();
3   JsonArray elements = rootObj.getAsJsonArray(<TagName>);
4   for (JsonElement element : elements) {
5     JsonObject elementObj = element.getAsJsonObject();
6
7     String name = elementObj.get("NAME").getAsString();
```

7 Threads

7.1 Begriffsdefinitionen

- **Synchronisation**: Koordination nebenläufiger Abläufe, Regelung des Zugriffs auf gemeinsame Ressourcen
- **Verklemmung** (Deadlock): Jeder Ablauf wartet auf Bedingung, die nur durch anderen (ebenfalls wartenden) Ablauf beseitigt werden kann
- **Aushungerung** (Starvation): Ablauf wird nie ausgeführt
- **Fairness**: Jedem Ablauf wird Ressource zugeteilt

7.1.1 Systeme

- **Sequentielles System**: Sequenz von Anweisungen
- **Nebenläufiges System** (Concurrent System): Ströme von Anweisungen, die parallel oder pseudoparallel ausgeführt werden
- **Verteiltes System** (Distributed System): räumliche bzw. konzeptionelle Aufteilung einzelner Komponenten eines Systems

7.1.2 Prozesse & Threads

- Prozesse
  - Eigene Ressourcen (Adressraum)
  - Verwaltung durch Betriebssystem
- Threads
  - läuft innerhalb eines Prozesses, hat Zugriff auf dessen Ressourcen
  - Verwaltung durch Benutzerprogramm (z.B. JVM)
  - Leichtgewichter Prozess (Lightweight Process)

7.1.3 Synchronisation

Synchronisationskonzepte: Monitore, Semaphore

- Gemeinsam genutzte Ressourcen minimieren
- Seiteneffektfrei programmieren
- Daten austauschen statt auf gemeinsame Daten zugreifen

7.2 Interfaces

- **Thread**: Klasse, die einen einzelnen Thread repräsentiert. Enthält Methoden wie *start, run, suspend, resume, interrupt, join, und terminate*
- **Runnable**: Interface, dass eine einzelne Methode *void run()* enthält
- **Executer**: Interface, dass Methoden zu Runnable Methoden enthält
- **Executorservice**: Interface, dass Executor erweitert und Methoden bereitstellt, um die Dauer eines Thread Pools zu bestimmen

- **Future**: Interface, dass das Ergebnis einer Asynchronen Berechnung darstellt
- **Callable**: Interface, dass eine *call()* Methode enthält, die das Ergebnis einer Berechnung zurückgibt und Exceptions werfen kann

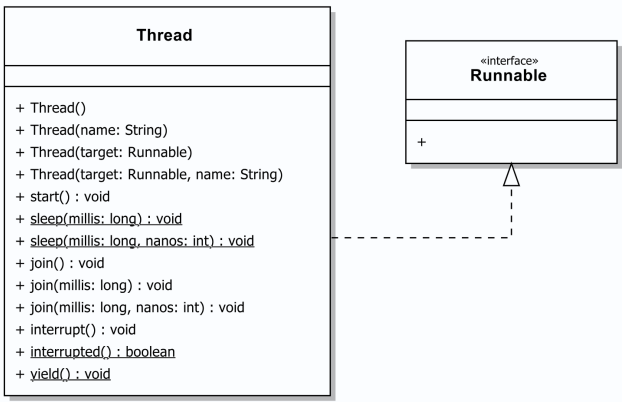

Figure 8: Runnable Interface & Thread Class UMLs